JOURNAL OF COMPUTER AND SYSTEM SCIENCES 9, 332–354 (1974)

Correct and Optimal Implementations of Recursion in a Simple Programming Language*

JEAN VUILLEMIN

Iria-Laboria, 78-Rocquencourt, France

Received July 16, 1973

The object of this paper is to study the mechanism of recursion in a simple, LISP-like programming language, where the only means of iteration is through recursion. The theory of computation developed in Scott [6] provides the framework of our study. We show how the implementations of recursion which deserve to be called "correct" can be characterized semantically, and demonstrate a general criterion for the correctness of an implementation. We then describe an implementation of recursion which is both correct and optimal in a general class of sequential languages, and therefore constitutes an attractive alternative to both "call-by-name" and "call-by-value."

INTRODUCTION

It is more or less the programmer's intuition that, unless side-effects take place, the result of a recursive routine, if any, will not depend on whether its parameters have been declared as name or value parameters. It would also seem natural that "call-by-value" be more efficient than "call-by-name," since it avoids many redundant computations of arguments. As pointed out by Morris [4] and Cadiou [1], this is not always the case, and there are recursive routines which never terminate when using "call-by-value" and yield their result quite rapidly with "call-by-name."

The purpose of this paper is to study the functions computed by various implementations of recursion. Unlike Cadiou [1] who mostly studied the functions computed by "call-by-value," we devote our energy to characterizing implementations equivalent to "call-by-name," which deserve the name of *correct implementation* because they terminate whenever it is possible to do so. We then describe a correct implementation of recursion, the *delay rule* which is optimal in that it computes its result faster than any other rule for any argument.

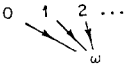
Some Minimal Background in Scott's Theory of Computation

A *data-type* is the semantic interpretation of the domain on which a computation is performed. *Abstractly, a data-type is a set partially ordered by the relation* $\subseteq$ *in which*

* Part of this research was supported by the Department of Computer Science at Stanford University and Stanford Artificial Intelligence Project.

any chain $x_0 \subseteq \cdots \subseteq x_n \subseteq x_{n+1} \cdots$ *has the least-upper bound or limit* $\bigcup_{n\geqslant 0} x_n$ *such that* $x_n \subseteq \bigcup_{i\geqslant 0} x_i$ *for any* n *and* $x_n \subseteq x$ *for any* n *implies* $\bigcup_{n\geqslant 0} x_n \subseteq x$.

The relation $x \subseteq y$ means that x is an approximation to y, and a data-type $\mathscr{D}$ has a least element $uu_{\mathscr{D}}$ such that $uu_{\mathscr{D}} \subseteq x$ for any $x \in \mathscr{D}$, which represents a complete absence of information. For example, the data-type N of natural numbers can be represented by



where ω is an abbreviation for uu_N.

In this case, $x \subseteq y$ means either that x is *undefined*, i.e., is the result of some non-terminating procedure, or that x and y are the same integer.

The data-type $[\mathscr{D} \to \mathscr{D}]$ of functions over some data-type $\mathscr{D}$ can be ordered by $f \subseteq g$ in $\mathscr{D} \to \mathscr{D}$ whenever $f(x) \subseteq g(x)$ in $\mathscr{D}$ for all $x \in \mathscr{D}$. In $[N \to N]$, let us define the function *one* as *one* $(x) \equiv 1$ for any natural number x and *one* $(\omega) \equiv \omega$. (Here, $x \equiv y$ means $x \subseteq y$ and $y \subseteq x$). This function can also be defined as the limit of a chain of partial functions, one $\equiv \bigcup_{i\geqslant 0} [\lambda x$ *if* $x < i$ *then* 1 *else* $\omega]$.

If we now consider programs as functions mapping data-types, *computable functions ought to be monotone and continuous, i.e.,* $x \subseteq y$ *implies* $f(x) \subseteq f(y)$ *and* $x \equiv \bigcup_{i\geqslant 0} x_i$ *implies* $f(\bigcup_{i\geqslant 0} x_i) \equiv \bigcup_{i\geqslant 0} f(x_i)$ (see Scott [6] for motivation). For functions of several arguments, continuity and monotonicity are meant componentwise.

One can then formulate Kleene's first recursion theorem as follows:

THEOREM 1. *Any continuous function* $\lambda x \cdot f(x)$ *over a data-type* $\mathscr{D}$ *has a least-fixed-point* x_f *and* $x_f \equiv \bigcup_{n\geqslant 0} f(uu_{\mathscr{D}})$.

Proof. Here $f^n(uu)$ means $f(f(\cdots(f(uu)\cdots)$ (n times) and, by monotonicity of f, the sequence $uu \subseteq f(uu) \subseteq \cdots \subseteq f^n(uu) \subseteq \cdots$ is indeed a chain. By continuity $f(x_f) \equiv f(\bigcup_{n\geqslant 0} f^n(uu)) \equiv \bigcup_{n\geqslant 0} f^{n+1}(uu) \equiv x_f$ and x_f is indeed a fixed-point of f. Let y be an arbitrary fixed-point of f, i.e., $y \equiv f(y)$. It is easy to prove by induction that $f^n(uu) \subseteq y$ for any n. The conclusion $x_f \equiv \bigcup_{n\geqslant 0} f^n(uu) \subseteq y$ follows by definition of limits, and x_f is indeed minimal. ∎

1. COMPUTATIONS OF RECURSIVELY DEFINED FUNCTIONS

Before defining a computation rule, we describe two programming languages, *lang S* and *lang P*. Although those two languages were chosen for their extreme simplicity, their use of recursion is a general as any, and the results of this paper provide some insight into semantics and implementation of more complex programming languages.

Lang S permits only *sequential* computations, and corresponds to a certain "typed" subset of Algol or LISP.

Lang P requires some parallel operations, and thus departs from more classical programming languages, although we could undoubtedly write an interpreter for lang *P* in any of those classical languages.

1.1. *Description of lang S and lang P*

Syntax

Both languages have the same syntax:

$$\begin{aligned}
\langle\text{program}\rangle: :=\ & F(X_1,\ldots,X_n) \Leftarrow \langle\text{term}\rangle \\
\langle\text{term}\rangle: :=\ & A_1 \mid A_2 \mid \cdots \\
& \mid X_1 \mid \cdots \mid X_n \\
& \mid G_1(\langle\text{term } 1\rangle,\ldots,\langle\text{term } p_1\rangle) \\
& \vdots \\
& \mid G_k(\langle\text{term } 1\rangle,\ldots,\langle\text{term } p_k\rangle), \\
& \mid F(\langle\text{term } 1\rangle,\ldots,\langle\text{term } n\rangle).
\end{aligned}$$

We limit ourselves to a single recursive equation, the extension of the results in this paper to systems of mutually recursive equations being straightforward.

Here, $A_1, A_2,\ldots, G_1,\ldots, G_k$ denote fixed constants and functions, respectively. It is convenient to use a more standard syntax, e.g., $F(X) \Leftarrow$ IF $X = 0$ THEN 1 ELSE $X \cdot F(X - 1)$ instead of $F(X) \Leftarrow G_1(P_1(X, A_0), A_1, G_2(X, F(G_3(X, A_1))))$.

The meaning of a program will be a continuous mapping in $[\mathscr{D}_1 x \cdots x\mathscr{D}_n \to \mathscr{D}]$, where each $\mathscr{D}_i$ and $\mathscr{D}$ are some data-types; for simplicity, the $\mathscr{D}_i$'s will be identical to $\mathscr{D}$ unless explicitly specified. The meaning of a $\langle$term$\rangle$ is a (continuous) functional $\lambda f \cdot \lambda x_1,\ldots, x_n\ \mathscr{S}\ [\langle\text{term}\rangle]$ where the semantic function $\mathscr{S}[T](f)(x_1,\ldots, x_n)$ or $\mathscr{S}[T](\zeta)$ for short is defined inductively as follows:

(i) $\mathscr{S}[A_i](\zeta) \equiv a_i$, where $a_i \in \mathscr{D}$

(ii) $\mathscr{S}[X_i](\zeta) \equiv x_i$

(iii) $\mathscr{S}[G_k(\langle\text{term } 1\rangle,\ldots,\langle\text{term } p_k\rangle](\zeta) \equiv g_k(\mathscr{S}[\langle\text{term } 1\rangle](\zeta),\ldots, \mathscr{S}[\langle\text{term } p_k\rangle](\zeta))$ where g_k is some continuous function in $[\mathscr{D}^{p_k} \to \mathscr{D}]$.

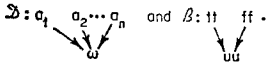
(iv) $\mathscr{S}[F(\langle\text{term } 1\rangle,\ldots, \langle\text{term } n\rangle)](\zeta) \equiv f(\mathscr{S}[\langle\text{term } 1\rangle](\zeta),\ldots, \mathscr{S}[\langle\text{term } n\rangle](\zeta))$.

Semantics of Terms in lang S.

The semantics of lang *S* is defined in precisely the same way, the difference lying in restrictions on the interpretation of base functions. In lang *S*, we require functions to be sequential, i.e., roughly that their arguments can be computed in sequence. We shall

give later a precise definition of this notion. For expository purposes, however, we shall limit ourselves for the moment to studying a particular sequential language.

The data-types on which our particular lang S is computing are *discrete*, i.e., $(x \subseteq y)$ iff $(x = \omega$ or $x = y)$. Typical examples are:



In what follows, we use ω instead of $uu_{\mathscr{D}}$ and Ω in place of $uu_{\mathscr{D}\to\mathscr{D}}$ in order to help the eye avoid type confusions. Among the base functions, we point out a particular one, denoted if-then-else whose interpretation is the usual conditional, i.e., *if uu then x else* $y \equiv \omega$, *if tt then x else* $y \equiv x$ and *if ff then x else* $y \equiv y$.

All other base functions are required to be *strict*, i.e., $g_i(..., \omega, ...) \equiv \omega$: they are undefined as soon as (at least) one of their arguments becomes undefined. They are meant to correspond to the "hardware" functions: add, addone, test-for-equality,... .

It will be shown that all functions definable in lang S are sequential. The symmetric *OR* defined by the table:

x or y

x \ y	*uu*	*tt*	*ff*
uu	*uu*	*tt*	*uu*
tt	*tt*	*tt*	*tt*
ff	*uu*	*tt*	*ff*

or the symmetric multiply $*$ where $0 * x \equiv x * 0 \equiv 0$ are not sequential, and are therefore *not definable in lang S*, nor in Algol for that matter.

Semantics of Programs in both lang S and lang P.

The functional $\tau \equiv \lambda f \cdot \lambda x_1, ..., x_n \mathscr{S}[\langle \text{term} \rangle](\zeta)$ can be shown to be continuous. It must therefore have a least fixed-point f_τ and it would be nice to define the meaning $\mathscr{M}$ of the corresponding program as $\mathscr{M}[\langle \text{program} \rangle] \equiv f_\tau$. This is unfortunately not true for all implementations of recursion, and our goal will be to characterize the implementations for which the computed function is equal to this least fixed-point.

1.2. *Conventions and Notations*

The reader has already noticed that *syntactic entities are denoted by upper case letters, while the associated semantic objects are represented by the corresponding lower-case letters.* We shall keep this convention throughout this paper. For example, if T is IF $X = 0$

THEN 1 ELSE $X \cdot F(X-1)$, then $t \equiv \lambda f \cdot \lambda x$ *if* $x = 0$ *then* 1 *else* $x \cdot f(x-1)$, where $=$ in this last expression means the equality function over the natural numbers, 0 the number 0, etc.

From now on, we use upper case letters other than A, X, F, and G to denote (syntactic) terms. If T and S are terms, we denote by $T\{S/X_i\}$ the result of replacing all occurrences of the letter X_i by the term S in T. By $T\{P/F\}$, we mean the term obtained by replacing in T all subterms of the form $F(T_1, \ldots, T_n)$ by $P\{T_1/X_1, \ldots, T_n/X_n\}$, for example, if $T = G_1(F(X_1, F(X_1, X_2)), X_1)$ and $P = G(F(X_2, X_1))$ then $T\{P/F\} = G_1(G(F(G(F(X_2, X_1)), X_1)), X_1)), X_1)$.

Whenever we only wish to substitute P for some occurrences of F in T, we rename, say F_1, the occurrences that we shall substitute and F_2 the others. The result of the substitutions is then $T\{P/F_1, F/F_2\}$. The same kind of notation also applies to semantic terms. We use $F(\bar{X})$ and $f(\bar{x})$ as abbreviations for $F(X_1, \ldots, X_n)$ and $f(x_1, \ldots, x_n)$, respectively. Also, it will be convenient to only consider programs $F(\bar{X}) \Leftarrow P$, where P is of the form $G(P_1, \ldots, P_p)$ with the additional restriction that each of the letters F, $X_1, \ldots, X_n$ occurs at least once in P. That is, P is required not to ignore any of its program variables, to depend upon F (i.e., to be recursive) and not to be of the uninteresting form $F(\bar{X}) \Leftarrow F(T_1, \ldots, T_n)$. The main results of this paper generalize without this restriction (see [7]), but the proofs are made longer by an addition of special cases.

1.3. *Computation Rule*

A computation rule φ is an algorithm for selecting some occurrences of the letter F in each term. For any such rule and input $\bar{D}$, we construct the computation sequence $T_0, T_1, \ldots, T_n, \ldots$, of the term T by the program $F(\bar{X}) \Leftarrow P$ as follows $T_0 = T\{\bar{D}/\bar{X}\}$ and T_{i+1} is the result of substituting P for the F's chosen by φ in T_i. For example, if $P =$ IF $X < 2$ THEN X ELSE $F(X-1) + F(X-2)$, the computation sequence of $F(2)$ according to "call-by-value" is

$T_0 = \underline{F}(2)$,

$T_1 =$ IF $2 < 2$ THEN 2 ELSE $\underline{F}(1) + F(0)$,

$T_2 =$ IF $2 < 2$ THEN 2 ELSE IF $1 < 2$ THEN $1 + \underline{F}(0)$ ELSE $F(0) + F(-1) + F(0)$,

$T_3 =$ IF $2 < 2$ THEN 2 ELSE IF $1 < 2$ THEN IF $0 < 2$ THEN 1
ELSE $1 + F(-1) + F(-2)$, ELSE $F(0) + F(-1) + F(0)$,

$T_4 = T_5 = \cdots = T_3$.

(Here, $F(1)$ is in fact an abbreviation for, say $F(A_2 - A_1)$, etc.).

In T_n, we underline the F's selected by the computation rule for substitution. It is interesting to see precisely how the underlined F is selected in this last example. For this purpose, we introduce the notion of simplification.

In its most general form, simplification can be an extremely powerful computation tool. For example, if our program is $F(X) \Leftarrow$ IF $X = 0$ THEN 0 ELSE $F(X - 1)$ it is perfectly all right to use $F(X) \rightarrow 0$ as a simplification rule over the natural numbers, and there is no room left for substitutions! Our purpose however is to study computations which are performed by substitutions and *not* by simplifications. We must therefore restrict the power of simplifications which we allow. At this point, we could merely use the notion of *standard simplification*, as defined by Cadiou [1]. Roughly, standard simplifications force us to know everything about base functions, and nothing a priori about the recursively defined function F, since simplifications of the type $F(\bar{D}) \rightarrow A_i$ are not permitted. In effect, we have to compute without any "built in" value of the recursively defined function, stored for example in memory from a previous computation.

However, we shall use the less restrictive notion of canonical simplifications. A set of simplification rules is *canonical* if it has the following properties.

(i) Each term T can be simplified into a unique term simpl(T) which cannot be simplified any further.

(ii) The simplifications are consistent with the interpretation, i.e., simpl(T) = simpl(T') implies $t(\Omega) \equiv t'(\Omega)$ for any terms T and T'.

(iii) The simplification set is complete, i.e., $t(\Omega) \equiv a_i$ implies simpl(T) = A_i for any term T and constant A_i.

Condition (i) means that the simplification process is finite and unambiguous; (ii) expresses that the semantic information contained in each term is not affected by simplifications; finally, (iii) can be regarded as meaning that, if a computation terminates, the simplifications can tell us so. A canonical set of simplification rules for the natural numbers is

$$A_i + A_j \rightarrow A_{i+j};$$

$$A_i - A_j \rightarrow A_{i-j} \quad \text{if} \quad i \geqslant j;$$

$$A_i < A_j \rightarrow \begin{cases} \text{TRUE} & \text{if} \quad i < j \\ \text{FALSE} & \text{if} \quad i \geqslant j; \end{cases}$$

$$\text{IF TRUE THEN } T \text{ ELSE } T' \rightarrow T,$$

$$\text{IF FALSE THEN } T \text{ ELSE } T' \rightarrow T'.$$

Using the same example as above, we have:

$$\begin{aligned} \text{simpl}(T_0) &= F(A_2); \\ \text{simpl}(T_1) &= F(A_1) + F(A_0); \\ \text{simpl}(T_2) &= A_1 + F(A_0); \\ \text{simpl}(T_3) &= \text{simpl}(T_4) = \cdots = A_1. \end{aligned}$$

We see that "call-by-value" selects the leftmost-innermost occurrence of F in simplified terms. Similarly, "call-by-name" selects the "leftmost-outermost" one.

The standard simplifications of Cadiou [1] can also be shown to be canonical. From now on, we always implicitly assume the existence of a canonical set of simplification rules, for Lang S or Lang P. It is occasionally described explicitly, when we have a specific interpretation of either language in mind.

1.4. *Computation lattice of a program*

Instead of considering computation sequences for each input and computation rule, we can apprehend the set of all possible computations in one infinite diagram. For example, the computation diagram of the term $F(F(X))$ by the program $F(X) \Leftarrow G(X, FF(X))$ looks like.

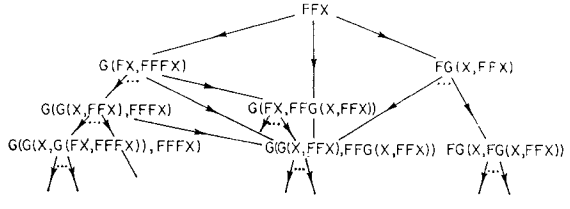


FIG. 1. Computation lattice of $F(F(X))$ by $F(X) \Leftarrow G(X, FFX)$.

A computation rule is then an algorithm for selecting a path in such a graph for each input. This computation diagram has a very rich structure which we shall study.

Computation of a Term According to P.

We say that $B \rightarrow_P C$ or simply $B \rightarrow C$ whenever C can be obtained by substituting P for some occurrences of F in B. The notation $B \overset{*}{\rightarrow}_P C$ or $B \overset{*}{\rightarrow} C$ means that there exists a finite sequence of terms $D_0, D_1, \ldots, D_m$ such that $D_0 = B$, $D_m = C$ and $D_i \rightarrow_P D_{i+1}$ for $0 \leqslant i < m$.

DEFINITION. The computation diagram of T by P is the set of terms U such that $T \overset{*}{\rightarrow}_P U$, partially ordered by $\leqslant$ where $B \leqslant C$ whenever $B \overset{*}{\rightarrow}_P C$. It is clear that $\leqslant$ is reflexive and transitive. In order to prove that it is also antisymmetric, we notice that, if $B \rightarrow_P C$, the size $\| C \|$ (where size is, say the number of symbols) of the term C is strictly larger than the size of B if at least one substitution has been performed (this is due to our restriction on P). It follows that $B \overset{*}{\rightarrow} C$ and $C \overset{*}{\rightarrow} B$ implies $B = C$.

Clearly, the computation diagram of T by P has the Church–Rosser property of the λ-calculus. (This follows from the work of Rosen [5] for example and a direct proof

is given in [7]). However, it also has a property which is *not* true of the λ-calculus, namely:

THEOREM 2. *The computation diagram of T by P is a lattice under the ordering $\leqslant$, and we name it the computation lattice of T by P.*

Proof. In order to study the structure of the computation diagram of a term T_0 by a program P, we need to relate the structure of C to that of B when $B \overset{*}{\to}_P C$.

LEMMA 1.

(i) $A_i \overset{*}{\to} C$ *if and only if* $C = A_i$ *and* $X_j \overset{*}{\to} C$ *if and only if* $C = X_j$.

(ii) $G_i(B_1, \ldots, B_{p_i}) \overset{*}{\to} C$ *if and only if* $C = G_i(C_1, \ldots, C_{p_i})$ *and* $B_i \overset{*}{\to} C_i$ *for* $1 \leqslant i \leqslant p_i$.

(iii) $F(B_1, \ldots, B_n) \overset{*}{\to} C$ *if and only if* $C = F(C_1, \ldots, C_n)$ *with* $B_i \overset{*}{\to} C_i$ *for* $1 \leqslant i \leqslant n$ *or* $P\{B_1/X_1, \ldots, B_n/X_n\} \overset{*}{\to} C$.

Proof. Claims (i) and (ii) are easy and we only prove (iii). If $B = F(B_1, \ldots, B_n) \overset{*}{\to} C$ and C is not of the form $F(C_1, \ldots, C_n)$, there must be a point in the computation $B \overset{*}{\to} C$ where the outermost F of B is substituted, i.e., $F(B_1, \ldots, B_n) \overset{*}{\to} F(B_1', \ldots, B_n') \to P\{B_1''/X_1, \ldots, B_n''/X_n\} \overset{*}{\to} C$ with $B_i' \to B_i''$ (and therefore $B_i \overset{*}{\to} B_i''$) for any $1 \leqslant i \leqslant n$. It follows from our definitions that $B_i \overset{*}{\to} B_i''$ for $1 \leqslant i \leqslant n$ implies $P\{B_1/X_1, \ldots, B_n/X_n\} \overset{*}{\to} P\{B_1''/X_1, \ldots, B_n''/X_n\}$ and consequently $P\{B_1/X_1, \ldots, B_n/X_n\} \overset{*}{\to} C$, as claimed in (iii). In order to get the other part of the implication (iii), we simply notice that

$$F(B_1, \ldots, B_n) \to P\{B_1/X_1, \ldots, B_n/X_n\}$$

by substituting P for the outer F in $F(B_1, \ldots, B_n)$. ∎

If $B \leqslant C$, we can define a distance dist(B, C) between B and C as follows.

(i) IF $B = A_i$ or $B = X_j$ then $C = B$ and $\text{dist}(B, C) = 0$;

(ii) if $B = G_i(B_1, \ldots, B_{p_i})$, then $C = G_i(C_1, \ldots, C_{p_i})$ with $B_i \leqslant C_i$ for $1 \leqslant i \leqslant p_i$ and $\text{dist}(B, C) = \max_{1 \leqslant j \leqslant p_i}\{\text{dist}(B_j, C_j)\}$;

(iii) if $B = F(B_1, \ldots, B_n)$ then (by Lemma 1), either $C = F(C_1, \ldots, C_n)$ and $\text{dist}(B, C) = \max_{1 \leqslant i \leqslant n}\{\text{dist}(B_i, C_i)\}$ or $P\{B_1/X_1, \ldots, B_n/X_n\} \leqslant C$ and $\text{dist}(B, C) = 1 + \text{dist}(P\{B_1/X_1, \ldots, B_n/X_n\}, C)$.

It is easily seen (by induction on the length of the derivation $B \overset{*}{\to} C$) that the distance between any two terms $B \leqslant C$ is finite.

LEMMA 2. *If $B = F(B_1, \ldots, B_n)$, $C = F(C_1, \ldots, C_n)$, $B' = P\{B_1/X_1, \ldots, B_n/X_n\}$ and $C' = P\{C_1/X_1, \ldots, C_n/X_n\}$ then $B \leqslant C$ implies $B' \leqslant C'$ and $\text{dist}(B', C') \leqslant \text{dist}(B, C)$.*

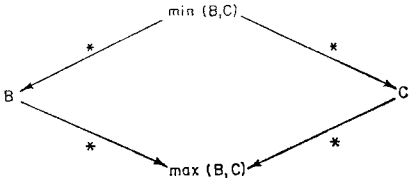
Proof. By a straightforward induction on $\| P \|$, one proves that

$$\text{dist}(P\{B_1/X_1, \ldots, B_n/X_n\}, P\{C_1/X_1, \ldots, C_n/X_n\}) \leqslant \max_{1 \leqslant i \leqslant n} \{\text{dist}(B_i, C_i)\},$$

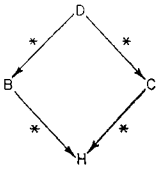
hence $\text{dist}(B', C') \leqslant \text{dist}(B, C)$. ∎

We now start the proof of Theorem 1:

For any two terms B, C in the computation diagram of T by P, we must show the existence of $\min(B, C)$ and $\max(B, C)$ such that



and for any D and H



implies $D \leqslant \min(B, C)$ and $\max(B, C) \leqslant H$.

Existence of $\max(B, C)$.

We shall describe an algorithm for computing $\max(B, C)$ and then prove the correctness of this algorithm: let $\sigma(B, C)$ be defined recursively as

(i) $\sigma(B, B) = B$,

(ii) $\sigma(G_i(B_1, \ldots, B_{p_i}), G_i(C_1, \ldots, C_{p_i})) = G_i(\sigma(B_1, C_1), \ldots, \sigma(B_{p_i}, C_{p_i}))$,

(iii) $\sigma(F(B_1, \ldots, B_n), F(C_1, \ldots, C_n)) = F(\sigma(B_1, C_1), \ldots, \sigma(B_n, C_n))$,

(iv) $\sigma(F(B_1, \ldots, B_n), G(C_1, \ldots, C_p)) = \sigma(P\{B_1/X_1, \ldots, B_n/X_n\}, G(C_1, \ldots, C_p)) = \sigma(G(C_1, \ldots, C_p), F(B_1, \ldots, B_n))$,

(v) in all the other cases, $\sigma(B, C)$ yields an error symbol, (say a german gothic letter) which is not part of our set of letters.

We shall prove that $\sigma(B, C) = \max(B, C)$ in two parts:

Part 1. *For any terms* T, B, C.

The proof is by induction on couples $\langle \mathrm{dist}(T, B) + \mathrm{dist}(T, C), \| T \| \rangle$ ordered lexicographically by $\prec$. Assuming the result to be true all for triples T', B', C' with $\langle \mathrm{dist}(T', B') + \mathrm{dist}(T', C'), \| T \| \rangle \prec \langle \mathrm{dist}(T, B) + \mathrm{dist}(T, C), \| T \| \rangle$, we prove if for T, B, C by a case analysis on the structure of T.

Case 1. $T = A_i$ or $T = X_j$.

By Lemma 1, $T \xrightarrow{*} B$ and $T \xrightarrow{*} C$ implies $T = B = C$; hence $B = C = \sigma(B, C)$ and indeed $B \xrightarrow{*} \sigma(B, C)$ and $C \xrightarrow{*} \sigma(B, C)$.

Case 2. $T = G_i(T_1, \ldots, T_{p_i})$.

By Lemma 1, $B = G_i(B_1, \ldots, B_{p_i})$ and $C = G_i(C_1, \ldots, C_{p_i})$, with $T_i \xrightarrow{*} C_i$ for $1 \leqslant i \leqslant p_i$. Since $\mathrm{dist}(T_i, B_i) + \mathrm{dist}(T_i, C_i) \leqslant \mathrm{dist}(T, B) + \mathrm{dist}(T, C)$ and $\| T_i \| < \| T \|$ for any $1 \leqslant i \leqslant p_i$, the induction hypothesis tells us that $B_i \xrightarrow{*} \sigma(B_i, C_i)$ and $C_i \xrightarrow{*} \sigma(B_i, C_i)$ for each $1 \leqslant i \leqslant p_i$. Regrouping everything, the conclusion $B \xrightarrow{*} \sigma(B, C)$ and $C \xrightarrow{*} \sigma(B, C)$ then follows from the definition $\sigma(C_i(B_1, \ldots, B_{p_i}), G_i(C_1, \ldots, C_{p_i})) = G_i(\sigma(B_1, C_1), \ldots, \sigma(B_{p_i}, C_{p_i}))$.

Case 3. $T = F(T_1, \ldots, T_n)$.

By symmetry, we only need consider the following subcases.

Case 3.1. $B = F(B_1, \ldots, B_n)$ and $C = F(C_1, \ldots, C_n)$.

The proof is similar to that of Case 2.

Case 3.2. $B = F(B_1, \ldots, B_n)$ and $C = G(C_1, \ldots, C_p)$.

Let $T' = P\{T_1/X_1, \ldots, T_n/X_n\}$ and $B' = P\{B_1/X_1, \ldots, B_n/X_n\}$. By Lemma 1, we know that $T' \xrightarrow{*} C$ and $T_i \xrightarrow{*} B_i$ for $1 \leqslant i \leqslant n$, hence $T' \xrightarrow{*} B'$. By Lemma 2, we know that $\mathrm{dist}(T', B') \leqslant \mathrm{dist}(T, B)$. Since $\mathrm{dist}(T', C) < \mathrm{dist}(T, C)$, we can apply the induction hypothesis to the terms T', B', C', i.e., $B' \xrightarrow{*} \sigma(B', C)$ and $C \xrightarrow{*} \sigma(B', C)$. Since $B \to B'$ and $\sigma(B, C) = \sigma(B', C)$ by definition of σ, we have established that $B \xrightarrow{*} \sigma(B, C)$ and $C \xrightarrow{*} \sigma(B, C)$.

Case 3.3. $B = G(B_1, \ldots, B_p)$ and $C = G(C_1, \ldots, C_p)$. Let $T' = P\{T_1/X_1, \ldots, T_n/X_n\}$. By Lemma 1, we know that $T' \xrightarrow{*} B$ and $T' \xrightarrow{*} C$. Since $\mathrm{dist}(T', C) < \mathrm{dist}(T, B)$ and $\mathrm{dist}(T', C) < \mathrm{dist}(T, C)$, we can use the induction hypothesis in order to get $B \xrightarrow{*} \sigma(B, C)$ and $C \xrightarrow{*} \sigma(B, C)$.

Part 2. *For any terms B, C, D.*

implies $\sigma(B, C) \leqslant D$.

The proof is by induction on $\langle \mathrm{dist}(B, D) + \mathrm{dist}(C, D), \| D \| \rangle$.

Case 1. $D = A_i$ or $D = X_j$. Then $D = B = C = \sigma(C, B)$ and $\sigma(B, C) \xrightarrow{*} D$.

Case 2. $D = F(D_1,..., D_n)$ or $D = G_i(D_1,..., D_{p_i})$ where G_i is not G. The proof goes mutatis-mutandis as that of Part 1, Case 2.

Case 3. $D = G(D_1,..., D_p)$. We only need consider the cases:

Case 3.1. $B = G(B_1,..., B_p)$ and $C = G(C_1,..., C_p)$. Back to Case 2.

Case 3.2. $B = F(B_1,..., B_n)$ and $C = G(C_1,..., C_p)$. Let $B' = P\{B_1/X_1,..., B_n/X_n\}$. Since $\text{dist}(B', C) < \text{dist}(B, D)$, we know by the induction hypothesis that $\sigma(B', D) = \sigma(B, C) \xrightarrow{*} D$.

Case 3.3. $B = F(B_1,..., B_n)$ and $C = F(C_1,..., C_n)$. Let $B' = P\{B_1/X_1,..., B_n/X_n\}$ and $C' = P\{C_1/X_1,..., C_n/X_n\}$. The induction hypothesis tells us that $\sigma(B', C') \xrightarrow{*} D$. One then proves by induction on $\| P \|$ that

$$\begin{aligned}\sigma(B', C') &= \sigma(P\{B_1/X_1,..., B_n/X_n\}, P\{C_1/X_1,..., C_n/X_n\}) \\ &= P\{\sigma(B_1, C_1)/X_1,..., \sigma(B_n, C_n)/X_n\}.\end{aligned}$$

We conclude the proof by noticing that $\sigma(B, C) \to \sigma(B', C')$ since $\sigma(B, C) = F(\sigma(B_1, C_1),..., \sigma(B_n, C_n)) \to P\{\sigma(B_1, C_1)/X_1,..., \sigma(B_n, C_n)/X_n\} = \sigma(B', C')$.

Existence of $\min(B, C)$.

For any terms B, C in the computation diagram of T by P the set $\{L \mid L \leqslant B, L \leqslant C\}$ of lower bounds of B and C is not empty because $T \leqslant B$ and $T \leqslant C$ and *it is finite*. We know from elementary lattice theory that, if any two elements in a partially ordered set have a least-upper-bound, any nonempty finite subset also has a least-upper-bound We then define $\min(B, C)$ as $\max\{L \mid L \leqslant B, L \leqslant C\}$ and verify easily that min has all the desired properties. ∎

Relation between the Computation Lattice and the Data-type of Continuous Functions over $\mathscr{D}$

In order to characterize computed partial functions in terms of the semantic interpretation of a given computation lattice, we notice that

LEMMA 3. *For any terms* B, C *in the computation lattice of* T *by* P, $B \leqslant C$ *implies* $b(\Omega) \subseteq c(\Omega)$.

Proof. The proof is straightforward by induction on $\| B \|$. If $B = A_i$ and $B = X_j$ then $B = C$ and $b(\Omega) \equiv c(\Omega)$. If $B = G_i(B_1,..., B_{p_i})$, then $C = G_i(C_1,..., C_{p_i})$ and we know by induction that $b_j(\Omega) \subseteq c_j(\Omega)$ for $1 \leqslant j \leqslant p_i$. Since $[\lambda x_1,..., x_{p_i} \cdot g_i(x_1,..., x_{p_i})]$ is monotone with respect to any of its arguments, $b(\Omega) \equiv g_i(b_1(\Omega),..., b_{p_i}(\Omega)) \subseteq g_i(c_1(\Omega),..., c_{p_i}(\Omega)) \equiv c(\Omega)$. Finally, if $B = F(B_1,..., B_n)$ then $b(\Omega) \equiv \Omega \subseteq c(\Omega)$. ∎

In particular, to any computation sequence $T_0 \to T_1 \to \cdots T_n \to T_{n+1} \to \cdots$ according to some rule φ and input $\bar{D}$, we associate the chain $t_0(\Omega)(\bar{d}) \subseteq t_1(\Omega)(\bar{d}) \subseteq$

$\cdots \subseteq t_n(\Omega)(\bar{d}) \subseteq t_{n+1}(\Omega)(\bar{d}) \subseteq \cdots$. *The corresponding computed partial function* φ_p *is therefore characterized as* $\varphi_p \equiv \lambda \bar{d} \cdot \bigcup_{n \geq 0} t_n(\Omega)(\bar{d})$. From these definitions follows an easy generalization of a theorem of Cadiou [1].

THEOREM 3. *Any fixed-point of the equation* $f \equiv p(f)$ *is an extension of any function computed by the program* $F(\bar{X}) \Leftarrow P$.

Proof. For any computation sequence $T_0, T_1, \ldots, T_n, \ldots$, where $T_0 = F(\bar{X})$ we can easily prove (see [7]) by induction that $t_i(\Omega) \subseteq p^i(\Omega) \equiv p(p(\cdots p(\Omega) \cdots)))$ (i times) for all natural numbers i. Since p is continuous, $f_p \equiv \bigcup_{i \geq 0} p^i(\Omega)$, hence $t_i(\Omega) \subseteq f_p$ for any i. It follows that $\varphi_p \equiv \bigcup_{i \geq 0} t_i(\Omega) \subseteq f_p$ and, since $f_p \subseteq f$ for any fixed-point f of p, the conclusion $\varphi_p \subseteq f$ holds. ∎

2. CORRECT IMPLEMENTATION OF RECURSION

In this section, we try to characterize the computation rules φ such that $\varphi_p \equiv f_p$ for any program $F(\bar{X}) \Leftarrow P$, called *fixed-point computation rules.*

Here are some computation rules we shall consider, both in lang S and lang P:

(1) *Call by value*: substitute for the leftmost-innermost occurrence of F after simplifications.

(2) *Call by name*: substitute for the leftmost-outermost occurrence of F after simplifications.

(3) *Parallel innermost*: substitute for the occurrences of F having all of their arguments free of F's.

(4) *Parallel outermost*: substitute for all the F's which do not occur in any argument of another F.

(5) *Free argument*: substitute for all the occurrences of F having at least one of their arguments free of F's after simplifications.

(6) *Full substitution*: substitute for all the occurrences of F.

2.1. *Incorrect Computation Rules*

PROPOSITION 1. *In lang P, the rules* (1), (2), *and* (3) *are incorrect.*

Proof. Consider the program $F(X, Y) \Leftarrow$ IF $X = 0$ THEN 0 ELSE $F(X + 1, F(X, Y)) * F(X - 1, F(X, Y))$ where $*$ is the parallel multiplication function $0 * x \equiv x * 0 \equiv 0$. As far as simplifications are concerned, this means adding $T * A_0 \to A_0$, $A_0 * T \to A_0$, and $A_i * A_j \to A_{ij}$ for each $i > 0$ and $j > 0$, to the rules described earlier for natural numbers. The least fixed-point over the integers (considered as a

discrete data-type) of the corresponding functional is the constant function 0. The computation of $F(1, 0)$ using (1), (2), or (3) is infinite. ▮

PROPOSITION 2 (Morris [4], Cadiou [1]). *In lang S the rules* (1) *and* (3) *are incorrect.*

Proof. Consider $F(X, Y) \Leftarrow$ IF $X = 0$ THEN 0 ELSE $F(X - 1, F(X, Y))$ over the same domain as in the previous example. The corresponding least fixed-point over the nonnegative integers is again the constant function 0 while the computation of $F(1, 0)$ using rules (1) or (3) is infinite. ▮

2.2. *Safe Computation Rules*

We now define the class of safe computation rules, and show that they correspond to "correct" implementations of recursion. Let φ be a computation rule and B an arbitrary term in the computation lattice of T by P. In order to describe the effect of φ on B, we rename F_1 the occurrences of F selected for substitution by φ in B for some input $\bar{D}$, and F_2 the others.

DEFINITION. We say that φ is a safe computation rule if, for any term $B\{F/F_1, F/F_2\}$ in the computation lattice of T by P and for any input $\bar{D}$, $b\{\Omega/f_1, f_p/f_2\}(\bar{d}) \equiv b\{\Omega/f_1, \Omega/f_2\}(\bar{d})$. Intuitively, the computation is safe if the values of the F's which are not substituted (renamed F_2) are insufficient: As long as more information is not obtained about the other arguments (the F_1's), the information about B cannot be improved.

In order to clarify this definition, let us prove the safeness of some of our computation rules.

PROPOSITION 3. *In lang S, the rules* (2), *i.e., call-by-name and* (5), *i.e., free argument are safe.*

Proof. By induction on $\| C \|$ where $C = \text{simpl}(B)$: we first notice that, because of the semantic definition of lang S, if F occurs in C then $c(\Omega)(\bar{d}) \equiv \omega$ (remember that C has been simplified).

Case $C = A_i$. Any rule is safe in this case.

Case $C = G_i(C_1, \ldots, C_{p_i})$. If the letter F does not occur in C, any rule is safe. Otherwise, since both rules select at least one F on such terms, we know by our previous remark that $c\{\Omega/f_1, f_p/f_2\}(\bar{d}) \equiv \omega \equiv c\{\Omega/f_1, \Omega/f_2\}(\bar{d})$.

Case $C = F(C_1, \ldots, C_n)$. The safeness of rule (2) is straightforward since the outermost F is substituted. For the same reason, rule (5) is safe if at least one of the C_i is constant. If none of the C_i's is constant, then $c_i\{\Omega/f_1, f_p/f_2\}(\bar{d}) \equiv \omega$ for $1 \leqslant i \leqslant n$

and we must prove that $f_p(\omega,..., \omega) \equiv \omega$. This can be ensured by imposing in lang S the additional restriction that all program variables $X_1,..., X_n$ occur in simpl(P) hence $f_p(\omega,..., \omega) \equiv p(f_p)(\omega,..., \omega) \equiv \omega$. ▮

PROPOSITION 4. *The rules* (4), *i.e., parallel outermost and* (6), *i.e., full substitution are safe in both lang S and lang P.*

Proof. By induction on $\| B \|$.

Case $B = A_i$. Any rule is safe.

Case $B = G_i(B_1,..., B_{p_i})$. By induction, $b_i\{\Omega/f_1, f_p/f_2\}(\bar{d}) \equiv b_i\{\Omega/f_1, f_2\}(\bar{d})$ for $1 \leqslant i \leqslant p$ in both cases, hence safeness is also satisfied on b.

Case $B = F(B_1,..., B_n)$. Both rules select the outermost F hence $b\{\Omega/f_1, f_p/f_2\}(\bar{d}) \equiv \omega \equiv b\{\Omega/f_1, \Omega/f_2\}(\bar{d})$. ▮

Note that the computation rules that we already recognized as incorrect are all unsafe. In order to prove that safe rules are correct, we need to establish some technical lemmas. First of all, we determine some properties of the *min* of two terms:

LEMMA 4.

(i) $\min(G_i(B_1,..., B_{p_i}), G_i(C_1,..., C_{p_i})) = G_i(\min(B_1, C_1),..., \min(B_{p_i}, C_{p_i}))$.

(ii) $\min(P\{B_1/X_1,..., B_n/X_n\}, G(C_1,..., C_p)) = P\{M_1/X_1,..., M_n/X_n\}$, *where* $M_1,..., M_n$ *are such that* $F(M_1,..., M_n) = \min(F(B_1,..., B_n), G(C_1,..., C_p))$.

Proof. Property (i) is easy and property (ii) follows from the fact that

$$P\{M_1/X_1,..., M_n/X_n\} \overset{*}{\to} M' \overset{*}{\to} P\{B_1/X_1,..., B_n/X_n\}$$

with $M_i \overset{*}{\to} B_i$ for $1 \leqslant i \leqslant n$ implies that $M' = P\{M_1'/X_1,..., M_n'/X_n\}$, where $M_i \overset{*}{\to} M_i' \overset{*}{\to} B_i$ for $1 \leqslant i \leqslant n$. Again, an explicit proof can be found in [7]. ▮

Our next step is to prove that

LEMMA 5. *If* $B \to C$, *so that* $C = B\{P/F_1, F/F_2\}$ *after renaming the F's in* B_i *and* $\min(B, D) = \min(C, D)$ *then* $D \leqslant B\{F/F_1, P^m/F_2\}$ *for some natural number m.*

Proof. Here P^m means $P\{P^{m-1}/F\}$ for $m > 0$ and $P^0 = F(X_1,..., X_n)$. The proof is by induction on $\langle \text{dist}(M, B) + \text{dist}(M, D), \| M \| \rangle$, where $M = \min(B, D) = \min(C, D)$.

Case $M = A_i$ *or* $M = X_j$. In this case, $M = B = C = D$ and we can choose $m = 0$.

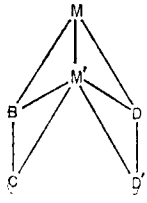
Case $M = G_i(M_1, ..., M_{p_i})$. By Lemma 1, $B = G_i(B_1, ..., B_{p_i})$, $C = G_i(C_1, ..., C_{p_i})$ and $D = G(D_1, ..., D_{p_i})$. By Lemma 3, $M_i = \min(B_i, D_i) = \min(C_i, D_i)$ for $1 \leqslant i \leqslant p$. It follows by induction that $D_i \leqslant B_i\{F/F_1, P^{m_i}/F_2\}$. We can then choose $m = \sup_{1 \leqslant i \leqslant p_i} \{m_i\}$ in order to get $D \leqslant B\{F/F_1, P^m/F_2\}$.

Case $M = F(M_1, ..., M_n)$. By definition of min, we need only consider the cases:

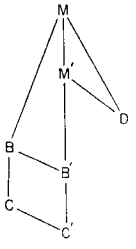
Case $B = F(B_1, ..., B_n)$ *and* $D = F(D_1, ..., D_n)$.

Subcase $C = F(C_1, ..., C_n)$. This subcase is treated exactly like the previous case.

Subcase $C = G(C_1, ..., C_p)$. By Lemma 4, there exists $C' = F(C_1', ..., C_n')$ such that $C = P\{C_1'/X_1, ..., C_n'/X_n\}$ and $\min(C, D) = \min(C', D)$, and we are back to the previous subcase.



Case $B = G(B_1, ..., B_p)$ *and* $D = F(D_1, ..., D_n)$. Let $M' = P\{M_1/X_1, ..., M_n/X_n\}$ and $D' = P\{D_1/X_1, ..., D_n/X_n\}$. By Lemma 3 $M' = \min(B, D') = \min(C, D')$. By Lemma 2, $\text{dist}(M', B) + \text{dist}(M', D') < \text{dist}(M, B) + \text{dist}(M, D)$ so we know by induction that $D' \leqslant B\{F/F_1, P^m/F_2\}$ and, a fortiori $D \leqslant B\{F/F_1, P^m/F_2\}$ for some m.



Case $B = F(B_1, ..., B_n)$ *and* $D = G(D_1, ..., D_p)$. Since $\min(B, D) = \min(C, D)$, the term C is also of the form $C = F(C_1, ..., C_n)$. Let $M' = P\{M_1/X_1, ..., M_n/X_n\}$, $B' = P\{B_1/X_1, ..., B_n/X_n\}$ and $C' = P\{C_1/X_1, ..., C_n/X_n\}$. By Lemma 3, we know that $M' = \min(B', D) = \min(C', D)$. By Lemma 2, $\text{dist}(M', B') + \text{dist}(M', D) < \text{dist}(M, B) + \text{dist}(M, D)$, and the induction hypothesis tells us that $D \leqslant B'\{F/F_1, P^m/F_2\}$. Since the outermost F has not been selected by φ in B, i.e., it has been labeled F_2, we have $B' \leqslant B\{P/F_2\}$. Our last case is then treated since $D \leqslant B\{F/F_1, P^{m+1}/F_2\}$. ∎

We now have the tools for proving that:

THEOREM 4. *Any safe rule is a fixed-point rule.*

Proof. In the computation lattice of $T_0 = F(\bar{D})$ by P, let $T_0, T_1, \ldots, T_n, \ldots$, and $S_0, S_1, \ldots, S_n, \ldots$ (where $S_0 = T_0$) be the computation sequences corresponding to respectively some safe rule φ and the full substitution rule. Since $s_n(\Omega) \equiv p^n(\Omega)$ then $\bigcup_{n\geqslant 0} s_n(\Omega) \equiv \bigcup_{n\geqslant 0} p^n(\Omega) \equiv f_p$. We know by Theorem 2 that $\varphi_p(\bar{d}) \subseteq f_p(\bar{d})$ and it is therefore sufficient to show that $\bigcup_{n\geqslant 0} s_n(\Omega)(\bar{d}) \subseteq \bigcup_{n\geqslant 0} t_n(\Omega)(\bar{d})$; in order to prove $\varphi_p \equiv f_p$. Let S_n be an arbitrary term in $S_0, S_1, \ldots$. Since there are only finitely many minorants of S_n in the computation lattice, there exists some j such that $\min(T_j, S_n) = \min(T_{j+1}, S_n)$. By Lemma 5, this implies $T_{j+1} = T_j\{P/F_1, F/F_2\}$ and $S_n \leqslant T_j\{F/F_1, P^m/F_2\}$ for m large enough. Lemma 3 tells us that $s_n(\Omega) \subseteq t_j\{\Omega/F_1, p^m(\Omega)/F_2\}$, hence $s_n(\Omega) \subseteq t_j\{\Omega/F_1, f_p/F_2\}$ since $p^m(\Omega) \subseteq f_p$. The rule φ being safe, $t_j\{\Omega/F_1, f_p/F_2\} \equiv t_j(\Omega)$ and therefore $s_n(\Omega)(\bar{d}) \subseteq t_j(\Omega)(\bar{d})$, hence $\bigcup_{n\geqslant 0} s_n(\Omega)(\bar{d}) \subseteq \bigcup_{m\geqslant 0} t_m(\Omega)(\bar{d})$. As a corollary, rules (2) and (5) are fixed-point in lang S and rules (4) and (6) are fixed-point rules in both lang S and lang P.

3. AN OPTIMAL IMPLEMENTATION OF RECURSION IN LANG S

Among the correct implementations of recursion, we now try to determine which ones are efficient. This proves unsuccessful in lang P, but we shall describe an implementation of recursion for lang S which turns out to be optimal.

We already know that, in lang S, "call-by-name" is a fixed-point rule, while "call-by-value" is not. However, "call-by-name" is not an efficient way of computing. For example, in the program $F(X) \Leftarrow$ IF $X > 0$ THEN $X - 1$ ELSE $F(F(X + 2))$ the "call-by-name" computation of $F(0)$ would be $F(0) \to F(F(2)) \to$ IF $F(2) > 0$ then $F(2) - 1$ ELSE $F(F(F(2) + 2)) \to F(2) - 1 \to 0$. What happens here is that the term $F(2)$ has been duplicated and subsequently computed twice. We shall describe a computation mechanism, called the *delay-rule*, which avoids those duplications, and prove its optimality.

3.1. *Never Do Today What You Can Put Off Until Tomorrow*

A natural way to keep track of duplications of terms is to assign labels to all occurrences of F in a computation sequence, so that copies of the same F will receive the same label. *This can be achieved by first labelling differently all F's in T_0 and P_0 then, if F is labelled α in T_n and is to be substituted, we label each occurrence of F after substitution by α followed by whatever labelling this particular occurrence had in P.* For example, using the same computation as before, and the labelling $F_1(X)$ for T_0 and IF $X > 0$

THEN $X-1$ ELSE $F_2(F_3(X+2))$ for P, the previous computation can be described as:

$$F_1(0) \to F_{12}(F_{13}(2)) \to \text{IF } F_{13}(2) > 0$$
$$\text{THEN } F_{13}(2) - 1 \text{ ELSE } F_{122}(F_{123}(F_{13}(2) + 2)) \to F_{13}(2) - 1 \to 0.$$

The whole idea of the delay-rule is to modify "call-by-name" so that, whenever some occurrence of F is substituted, all the occurrences having the same label will also be substituted. Hence, the "delay-rule" selects for substitution the leftmost-outermost F in a simplified term, as well as all the other F's having the same label.

Consequently, the delay rule computation of $F(0)$ in the program above is

$$F_1(0) \to F_{12}(F_{13}(2)) \to \text{IF } F_{13}(2) > 0 \text{ THEN } F_{13}(2) - 1 \text{ ELSE } F_{122}(F_{121}(F_{13}(2)+2)) \to 0.$$

At this point, it is clear that the "delay rule" is safe (proof similar to that of Proposition 1); what is not clear is that the "delay rule" should be more efficient than "call-by-name" and in fact, in our last example, it was less efficient since it took four substitutions versus three for "call-by-name" in order to obtain its result. When "call-by-name" computed $F_{13}(2)$ twice, the delay rule has been computing it three times! It is a simple exercise in data structuring however to avoid all those recomputations: instead of actually copying various occurrences of some F_α in a term, we simply set some pointers to a unique copy of the term F_α. Whenever any occurrence of F_α is chosen for substitution, the substitution is actually performed in the unique copy of F_α so that *all* occurrences of F_α are substituted at the price of one substitution.

Going a little bit away from our particular programming language we can sketch an implementation of this idea for, say Algol. The arguments of any procedure should be stored as pointers to formal expressions, together with a tag indicating that those arguments have not yet been computed. Whenever the value of an argument is explicitly needed, (for the evaluation of a conditional or on the right-hand side of an assignment), the tag is tested. If the value of the parameter is already there, we use it; otherwise the corresponding formal expression must be computed, its value kept for further references, and the tag is to be changed. In a machine like the Burroughs B5000 (see, for example, Lonergan–King [3]), the so-called "operand call syllabe" would do very nicely: depending on a tag stored with the operand, a load operation for example would get its argument either directly or through a subroutine call. Of course, one would then have to abandon "side-effects" altogether, (or change their intended semantics).

Before proving the optimality of the delay rule let us compare the efficiency of various computation rules on the programs

$$\text{Zer}(X) \Leftarrow \text{IF } X > 0 \text{ THEN } X - 1 \text{ ELSE Zer}(\text{Zer}(X + 2))$$
$$\text{Ack}(X, Y) \Leftarrow \text{IF } X = 0 \text{ THEN } Y + 1$$
$$\text{ELSE IF } Y = 0 \text{ THEN Ack}(X - 1, 1)$$
$$\text{ELSE Ack}(X - 1, \text{Ack}(X, Y - 1))$$

$\text{Ble}(X, Y) \Leftarrow \text{IF } X = 0 \text{ THEN } 1 \text{ ELSE } \text{Ble}(X - 1, \text{Ble}(X - Y, Y))$

$\text{Fib}(X) \Leftarrow \text{IF } X < 2 \text{ THEN } X \text{ ELSE } \text{Fib}(X - 1) + \text{Fib}(X - 2)$

over the integers.

	Zer(−2)	Ack(2, 1)	Ble(8, 2)	Fib(5)
Delay rule	7	14	9	15
Call by name	25	29	9	15
Call by value	7	14	341	15
Free argument	7	23	~4000	15
Full substitution[1]	11	23	~10000	15

[1] Strictly speaking, we are using the full substitution only on simplified terms, otherwise the computation would always be infinite.

The entries in this array indicate the number of substitutions required for computing the values at the top of the corresponding column, according to the rules at the left of the rows.

If he has been through those examples, the reader may feel quite disappointed because he can beat the delay-rule in almost all cases. For example, the hand-computation of Fib(5) only requires five substitutions if we are careful never to recompute an argument twice. It would be interesting to study a mechanism in which this type of computation would be possible; namely one could imagine a set of simplification rules which could be augmented dynamically, and allow some computations to be performed by simplifications of the style $F(\bar{D}) \to A$. In our scheme of things, however, this type of "built-in" values is not possible, since our only mean of computation is through substitutions, and *we should blame inefficiencies on the program, not on the computation rule.*

3.2. *Optimality of the Delay Rule*

So far, we know that the normal rule is safe, and that it never recomputes copies of the same term. Using the same labelling as before, we say that a label F_α is maximal in a term if α is not a proper initial segment of β for any label F_β in the term. *A term is simple if all of its labels are maximal.* In other words, a term is simple if all computations of various copies of subterms have been pushed to the same point. For example, if $P = F_2(F_1(X))$ and $T_0 = G(X, F_4(F_3(X)))$ then $G(G(X, F_{14}(F_{13}(X)), F_{24}(F_{23}(F_1(X)))))$ is not simple while $F_2(G(X, F_{14}(F_{13}(X))))$ is simple. *A computation is simple if all F's with the same labels are all treated alike in all substitutions* (if one of them is to be substituted, all of them are to be substituted). All terms in a simple computation

are necessarily simple. If we are to count for *one* a substitution of all F's with the same labels, as justified by our previous exercise in data structuring, simple computations are more efficient than others. Namely, if we define length($T_0 \overset{*}{\to} A$) as the total number of substitutions performed during the computation $T_0 \overset{*}{\to} A$, we have

LEMMA 6. *For any term A, there exists a simple term $\bar{A}$ with $A \leqslant \bar{A}$ such that, for any computation $T_0 \overset{*}{\to} A$ and simple computation $T_0 \overset{*}{\Rightarrow} \bar{A}$, length($T_0 \overset{*}{\Rightarrow} \bar{A}$) $\leqslant$ length($T_0 \overset{*}{\to} A$).*

Proof. Let $r(Q)$ be the number of maximal labels and $s(Q)$ be the sum of the lengths of the maximal labels in a term Q, while q and p mean respectively the number of occurrences of F in T_0 and P. It is easily proven by induction on length($T_0 \overset{*}{\to} Q$) that length($T_0 \overset{*}{\to} Q$) $\geqslant \varphi(Q, p, q)$ where $\varphi(Q, p, q) =$ if $p = 1$ then $s(Q)/q$ else $(r(Q) - q)/(p - 1)$. In a similar way, (Q simple) and ($T_0 \overset{*}{\Rightarrow} Q$ simple) imply length($T_0 \overset{*}{\Rightarrow} Q$) $= \varphi(Q, p, q)$.

Given any term A, we can "complete" it into an $\bar{A}$ by substituting P for all occurrences of F with non-maximal labels until there is none left. An $\bar{A}$ constructed in this way will be simple and such that $A \leqslant \bar{A}$ while $r(A) = r(\bar{A})$, $s(A) = s(\bar{A})$. It follows that, for any computation $T_0 \overset{*}{\to} A$ and simple computation $T_0 \overset{*}{\Rightarrow} \bar{A}$, length($T_0 \overset{*}{\Rightarrow} \bar{A}$) $= \varphi(\bar{A}, p, q) = \varphi(A, p, q) \leqslant$ length($T_0 \overset{*}{\Rightarrow} A$). ∎

The intuitive meaning of this lemma is very simple: nothing is to be gained by working on individual copies of the same term. At the same price, we get more information by substituting all copies of the same occurrences. In particular all the computation rules described so far will be improved by "lumping" together occurrences of F with the same labels, thus becoming simple rules. However they may still perform unnecessary substitutions unless:

THEOREM 5. *Any computation rule which is simple, safe and performs at most one substitution (using the data structuring indicated) at each computation step is optimal.*

Proof. Let T_0 be a term, $F(\bar{X}) \Leftarrow P$ a program and φ a safe and simple computation rule performing only one substitution at a time. Let $T_0 \Rightarrow T_1 \Rightarrow \cdots \Rightarrow T_n \Rightarrow T_{n+1} \Rightarrow \cdots$ the (simple) computation sequence of T_0 according to φ for some input $\bar{D}$.

If T is a term in the computation lattice of T_0 by P, let us consider an arbitrary computation $T_0 \overset{*}{\to}_{\mathscr{S}} T$, and prove that whatever approximation $t(\Omega)(\bar{d})$ of $t_0(f_p)(\bar{d})$ is computed by $\mathscr{S}$ will be computed *faster* by φ. For this purpose, we construct $\bar{T}$ as in Lemma E, and consider a simple computation $T_0 \overset{*}{\Rightarrow} \bar{T}$ (the argument in Lemma E not only proves the existence of $\bar{T}$ but also that of a simple computation $T_0 \overset{*}{\Rightarrow} \bar{T}$).

Let i be some natural number such that $T_i \leqslant T$ and $T_{i+1} \not\leqslant \bar{T}$. Since φ performs only one substitution at the time, this implies $T_i = \min(T_{i+1}, \bar{T}) = \min(T_i, \bar{T})$. Using the same argument as in the proof of Theorem 4, we then know that $\bar{t}(\Omega)(\bar{d}) \subseteq t_i(\Omega)(\bar{d})$. Using Lemmas 3 and 6 now, $T \leqslant \bar{T}$ implies $t(\Omega)(\bar{d}) \subseteq \bar{t}(\Omega)(\bar{d})$ and length

$(T_0 \overset{*}{\Rightarrow} \bar{T}) \leqslant \text{length}(T_0 \overset{*}{\Rightarrow} T)$. Since both $T_0 \overset{*}{\Rightarrow} \bar{T}$ and $T_0 \overset{*}{\Rightarrow} T_i$ are simple and $T_i \leqslant \bar{T}$, we have $\text{length}(T_0 \overset{*}{\Rightarrow} T_i) \leqslant \text{length}(T_0 \overset{*}{\Rightarrow} \bar{T})$ hence $t(\Omega)(d) \subseteq t_i(\Omega)(d)$ while $\text{length}\ (T_0 \overset{*}{\Rightarrow} T_i) \leqslant \text{length}(T_0 \overset{*}{\to}_{\mathscr{S}} T)$. ∎

We shall derive two applications of this theorem.

COROLLARY 1. *The delay rule is optimal in lang S.*

Proof. The delay rule has all the properties required by Theorem 5. ∎

COROLLARY 2. *In lang S, "call by value" is optimal whenever the least fixed-point f_p corresponding to the program $F(\bar{X}) \Leftarrow P$ is a strict function.*

Proof. Since "call by value" is clearly a simple rule and performs at most one substitution at each step, we only need proving that it is safe whenever f_p is strict. We prove that the substitution $B \to B'$ is safe in that case by induction on $\| C \|$ where $C = \text{simpl}(B)$:

Case $C = A_i$. Any rule is safe.

Case $C = G_i(C_1, \ldots, C_{p_i})$. Same argument as for the safeness of "call by name".

Case $C = F(C_1, \ldots, C_n)$. If F does not occur in any of the C_i's, then the outermost substitution is performed, which is clearly safe. Otherwise, let C_i be the leftmost term in which F occurs. Then, $C_i\{\Omega/f_1, f_p/f_2\}(d) \equiv \omega$ and $C\{\Omega/f_1, f_p/f_2\}(\bar{d}) \equiv f_p(\ldots, \omega, \ldots) \equiv \omega \equiv C\{\Omega/f_1, \Omega/f_2\}(\bar{d})$. ∎

3.3. *Sequential Functions*

The applications of Theorem 5 given in the previous section do not quite match with the generality of the result. In particular, the data-type on which lang S is computing are discrete. What we now sketch is a theory of sequential functions, where Theorem 5 finds its full application. The relevant notion here seems to be:

DEFINITION. A function $\lambda x_1, \ldots, x_n \cdot g(x_1, \ldots, x_n)$ in $[\mathscr{D}_1 x \cdots x \mathscr{D}_n \to \mathscr{D}]$ is sequential if, for all $x_1 \in \mathscr{D}_1, \ldots, x_n \in \mathscr{D}_n$ there exists an $i \in [1, n]$ such that, for all $y_1, \ldots, y_n$ such that $x_j \subseteq y_j$ for $j \in [1, n]$ and $x_i \equiv y_i$ we have $g(x_1, \ldots, x_n) \equiv g(y_1, \ldots, y_n)$.

Intuitively, g is sequential if, at any given moment, the value of (at least) one of its arguments is crucially needed in order to increase the value of the result. For the purpose of our theory, we need to check that sequentiality has the correct closure property, namely

PROPOSITION 5. *Sequentiality is preserved by composition of functions and fixed-point operators.*

Proof. Composition. If $\lambda z_1 ,..., z_n g(z_1 ,..., z_n)$ and $\lambda x_1 ,..., x_m f_i(x_1 ,..., x_m)$ for for $1 \leqslant i \leqslant n$ are sequential, then $\varphi \equiv \lambda x_1 ,..., x_m g(f_1(x_1 ,..., x_m),..., f_m(x_1 ,..., x_m))$ is also sequential: for any $x_1 ,..., x_m$ and $i \in [1, n]$, let $z_i \equiv f_i(x_1 ,..., x_n)$; since g is sequential $z_1 ,..., z_n$ determines some $i_0 \in [1, n]$ and, f_{i_0} being also sequential, $x_1 ,..., x_m$ determine some $j \in [1, m]$ which can then be used for the sequentiality of φ.

Fixed-point operator. If the functions $\lambda x_1 ,..., x_n f_i(x_1 ,..., x_n)$ are sequential for any natural number i, the function $\varphi \equiv \lambda x_1 ,..., x_n \bigcup_{i \geqslant 0} f_i(x_1 ,..., x_n)$ is also sequential: for any $x_1 ,..., x_n$ sequentiality of the f_i's determines a sequence $j_0 , j_1 ,...,$ where $j_i \in [1, n]$. At least one of the j_i's must occur infinitely often in thix sequence, and it can be used for proving that φ is sequential. ▮

For example, over a discrete data-type, conditional and strict functions are sequential; hence, by Proposition 5, all functions definable in lang S are sequential. In a data-type which is a lattice, the functions, $\lambda x, y$ sup(x, y) and $\lambda x \cdot y$ inf(x, y) are not sequential in general. The set Σ^ω of finite or infinite words over some vocabulary Σ becomes a data-type under the partial ordering: $x \subseteq y$ whenever x is an initial segment of y.

In Σ^ω, the functions

$\lambda x \cdot$ first(x) (take the first letter of x),

$\lambda x \cdot$ rest(x) (erase the first letter of x),

and

$\lambda x, y, x \otimes y$ (append the first letter of x to y) are sequential.

(The relevance of these functions and data-type to parallel programs is shown in Kahn [2].) This is clear enough for first and rest. For $x \otimes y$, if $x \equiv \Lambda$, i.e., x is the empty word, then the first argument is to be chosen for sequentiality because $\Lambda \otimes y \equiv \Lambda$; otherwise, $x \not\equiv \Lambda$ and any x' such that $x \subseteq x'$ will have the same first letter so that we can use the other argument y for sequentiality. The corresponding canonical set of simplification rules is: first$(A_i \otimes T) \rightarrow A_i$, rest$(A_i \otimes T) \rightarrow T$, $(A_i \otimes T) \otimes T' \rightarrow A_i \otimes T'$.

Yet another programming language. We define a new language *lang GS* similar to our previous ones except that all base functions must be sequential. Let $\mathscr{E}$ be a computation rule, called the generalized delay rule (GDR) defined as follows: First, using the same type of data-structuring as for the delay rule, $\mathscr{E}$ will be simple. In any term T, rule $\mathscr{E}$ will select at most one F (or rather set of F's with the same labels), as follows:

If $T = A_i$, no F is chosen.

If $T = G_i(T_1 ,..., T_{p_i})$, the F will be the F chosen by $\mathscr{E}$ in T_j where j is the indice corresponding to the sequentiality of g_i with the arguments $t_1(\Omega)(\vec{d}),..., t_{p_i}(\Omega)(\vec{d})$. Of course, this requires the choice of j to be effective; also, since we want $\mathscr{E}$ to be simple, all F's with the same labels occurring in other subterms are also to be substituted.

If $T = F(T_1, ..., T_n)$ the outermost F is selected by $\mathscr{E}$. We can apply Theorem 5 again in order to prove.

COROLLARY 3. *The generalized delay rule is optimal in lang GS.*

Proof. Since the GDR is simple and performs at most one substitution at each step, all we need to prove is that it is safe. The proof is by induction on $\| B \|$ where B is any term in the computation lattice of $T_0 = T\{\bar{D}/\bar{X}\}$ by P. The cases $B = A_i$ of $B = F(B_1, ..., B_n)$ are easy. If $B = G_i(B_1, ..., B_{p_i})$ and j is the sequentiality index of $g_i(b_1(\Omega)(\bar{d}), ..., b_{p_i}(\Omega)(\bar{d}))$, then $b_j\{\Omega/f_1, f_p/f_2\}(\bar{d}) \equiv b_j(\Omega)(\bar{d})$ by induction. Since $b_k(\Omega)(\bar{d}) \subseteq b_k\{\Omega/f_1, f_p/f_2\}(\bar{d})$, the very definition of sequentiality gives us

$$b\{\Omega/f_1, f_p/f_2\}(\bar{d}) \equiv b\{\Omega/f_1, \Omega/f_2\}(\bar{d}). \quad \blacksquare$$

For example, if the symbols f, r and $\otimes$ represent respectively the functions first, rest and append defined earlier, the GDR computation of $frFHAB$ (parenthesis omitted) with respect to the program $F(X) \Leftarrow X \otimes F(X)$, $H(X) \Leftarrow r(X) \otimes F(X)$ is $fr\ FHAB \to fr(HAB \otimes FHAB) \to fr((rAB \otimes FAB) \otimes F(rAB \otimes FAB)) \to fr((rAB \otimes FAB) \otimes (rAB \otimes FAB) \otimes F(rAB \otimes FAB))$. If we work only with simplified terms, the same computation becomes: $fr\,FHAB \to fr(HAB \otimes FHAB) \to fF(B \otimes FAB) \to B$. It turns out in this particular example that the GDR computation coincide with "call by value" (made into a simple rule), but this need not always be the case.

CONCLUSION

The results of this chapter should generalize quite nicely to a programming language where we introduce assignments, goto's and while statements. What is less clear to the author is how to perform computation in a "typeless" recursive language where procedures can be passed as arguments, say in a full LISP for example. For example, is there an optimal implementation of β-reduction in the λ-calculus? An efficient implementation, based upon an idea similar to the delay-rule, has been suggested by Wadsworth [8], but it is unfortunately not optimal. It might also be interesting to study (or prove the nonexistence of) optimal computation rules when the simplifications allowed are less restrictive than the ones we chose.

ACKNOWLEDGMENTS

We are grateful to Donald Knuth whose critism on an earlier draft led to many fruitful improvements. We are indebted to Jean Marie Cadiou for his contribution to the proof of Theorem 2, and to the referee for his careful reading and constructive criticism.

REFERENCES

1. J. M. CADIOU, Recursive definitions of partial functions and their computations, Ph.D. Thesis, Computer Science Department, Stanford University (1972).
2. G. KAHN, A preliminary theory of parallel programs, Rapport LABORIA, IRIA, 78-Rocquencourt, France (1973).
3. W. LONERGAN AND P. KING, Design of the B5000 system, *Datamation* **7** (1961), 28–32.
4. J. H. MORRIS, Lambda-Calculus Models of Programming Languages, Report MAC-TR-57, Mass. Inst. of Technology, 1968.
5. B. K. ROSEN, Tree-manipulating systems and Church–Rosser theorems, *J. Assoc. Comput. Mach.* **20** (1973), 160–187.
6. D. SCOTT, "Outline of a Mathematical Theory of Computation," Oxford Mono. PRG-2, Oxford University Press, Oxford, England, 1970.
7. J. VUILLEMIN, "Syntaxe, Sémantique et Axiomatique d'un langage de Programmation Simple," Thèse d'État, Rapport LABORIA IRIA, 78150 Rocquencourt, France (1974).
8. C. WADSWORTH, "Semantics and Pragmatics of the Lambda-Calculus," Ph.D. University of Oxford, Oxford, England, 1971.